Using Loops in Decision-Theoretic Refinement Planners

Richard Goodwin
rich@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA 15213-3890
Phone: (412) 268-8102 Fax: (412) 268-5576

Abstract

Classical AI planners use loops over subgoals to move a stack of blocks by repeatedly moving the top block. Probabilistic planners and reactive systems repeatedly try to pick up a block to increase the probability of success in an uncertain environment. These planners terminate a loop only when the goal is achieved or when the probability of success has reached some threshold. The tradeoff between the cost of repeating a loop and the expected benefit is ignored. Decision-theoretic refinement planners take this tradeoff into account, but to date, have been limited to considering only finite length plans. In this paper, we describe extensions to a decision-theoretic refinement planner, DRIPS, for handling loops. The extended planner terminates a loop when it can show that all plans with one or more additional iterations of the loop have lower utility. We give conditions under which optimal plans are finite and conditions under which the planner will find an optimal plan and terminate. With loops, a decision-theoretic refinement planner searches an infinite space of plans, making search control critical. We demonstrate how our sensitivity analysis-based search control technique provides effective search control without requiring the domain designer to hand-tune parameters, or otherwise provide search control information.

Introduction

Classical, STRIPS-like, AI planners use recursive sub-goals to, for example, move a stack of blocks by repeatedly moving the top block. Probabilistic planners, such as Weaver (Blythe 1994), Buridan (Kushmerick, Hanks, & Weld 1994) and ε-safe planners (Goldman & Boddy 1994), also repeat sequences of actions to achieve a goal with a given probability. Simple reactive systems use behavioural loops, for example, to repeatedly attempt to pick up a block until it is seen in the gripper (Musliner 1994). These planners terminate a loop only when the goal is achieved or when the probability of success has reached some threshold. The tradeoff between the expected cost of repeating a loop and the expected benefit is ignored. Decision-theoretic refinement planners take this tradeoff into account, but to date, have been limited to considering only finite length plans.

In this paper, we introduce extensions to a decision-theoretic planner, DRIPS (Haddawy & Doan 1994), that allow it to create plans with loops, specifically recursion, and to reason about how many iterations of a loop (recursive actions) are useful. We begin by introducing the basic planning mechanism used by DRIPS, a decision-theoretic refinement planner. We then give an example to illustrate our approach to encoding loops as recursions in the abstraction hierarchy. The resulting plan space is infinite, making completeness an issue. To ensure completeness, we give some conditions under which the optimal plan is finite and conditions under which the planner will find the optimal plan and terminate. We then turn our attention to search control, a crucial efficiency issue when dealing with large search spaces. We give empirical results using the DRIPS planner on a large medical domain and compare our sensitivity analysis-based search control method with hand-optimized control. Finally, we conclude with a discussion of future extensions to the DRIPS planner.

Decision-Theoretic Refinement Planning

A decision-theoretic refinement planner, such as DRIPS, takes as input a probability distribution over initial states, an action abstraction hierarchy and a utility function. It returns a plan, or set of plans, with the highest expected utility. The initial probability distribution gives the prior probabilities, such as the likelihood of a particular disease. The action abstraction hierarchy represents an abstraction of all possible plans in the domain. The actions can have conditional and probabilistic effects. The utility function maps chronicles of world state to utility values. A chronicle is a description of the state of the world over time(McDermott 1982). The planner evaluates a plan by projecting the (conditional) effects of its actions to create a set of chronicles. The expected utility of the plan is calculated by weighting the utility of each chronicle by its likelihood. Abstract actions in the plan can have ranges of attribute values and ranges of probabilities, that lead to a range of expected utility values. The planner begins with the top action in the abstraction hierarchy and refines it into a set of more detailed plans. These plans are evaluated and dominated plans are discarded. One plan dominates another if its range of expected utility is strictly greater than the other. The planner continues to refine actions in one of the non-dominated plans until all the non-dominated plans have been fully expanded. The remaining non-dominated plans are the set of

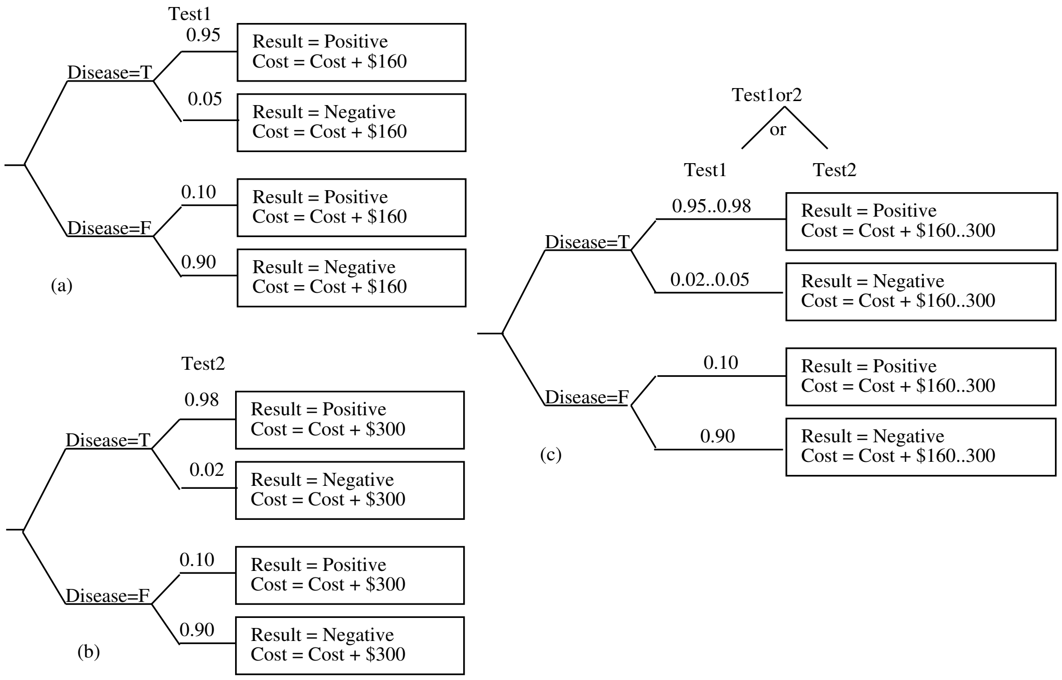


Figure 1: Test1 and Test2 are test actions that have different costs and accuracies. Test1or2 is an instance abstraction of the two tests.

optimal plans.

Action Abstraction

We focus now on the abstraction hierarchy used to encode the set of possible plans. Encoded in the hierarchy are instance abstractions and macro abstractions(Haddawy & Doan 1994). Instance abstractions are used to group together a set of actions with similar effects. Macro abstractions are used to summarize sequences of actions.

Consider a simple medical domain with two tests and a treatment for a disease. Suppose that one test is 95% accurate for detecting the presence of the disease and costs \$160 while the other test is 98% accurate but costs \$300. Both tests are 90% accurate for detecting the absence of the disease. Figures 1a and 1b give the definitions of the two tests and figure 1c gives their abstract instance action. The actions are conditional and probabilistic. The conditions, *disease* = *T* and *disease* = *F*, used to label the branches, are mutually exclusive and exhaustive. Each conditional branch can have probabilistic effects, given as a set of changes to attribute values labeled with their probability. Abstract actions can have effects with ranges of probabilities and ranges of attribute values (figure 1c).

One possible treatment plan is to test for the disease and treat if the test is positive. For our example, let the cost of treatment be \$5000, which cures the disease with 100% certainty and let the cost of an untreated case of the disease be \$100,000. Figure 2a gives the definition of the action that treats the disease if the test result is positive. The sequence of a test action followed by a Treat_if_Positive action is abstracted into the macro action given in figure 2b. The effects of the abstract action summarize the effects of the sequence, reducing the work needed to project the effects of the sequence. Fully expanding the Test_&_Treat_if_Positive action would give two plans, each with two actions. See (Haddawy & Doan 1994) and (Doan & Haddawy 1995a) for more details on the abstraction mechanism.

Comparing Treatment Plans

The planning objective for our simple medical domain is to create a testing and treatment policy that has the highest expected utility (lowest expected cost). One possible plan is to treat the disease unconditionally, which costs \$5000 per patient. Another possible plan is not to treat anyone. If the prior probability of the disease is 1/2, then the expected cost of not treating anyone is \$50,000. A third option is to use a test, and to treat if the test is positive. Using Test2 results in a plan that will treat most of the patients with the disease and not treat most of the healthy patients. The expected cost is \$4000. Using Test2 is better than always treating or never treating. However, the plan with Test2 will miss treating 2% of the patients with the disease and will treat 10% of the healthy patients.

Now suppose we repeat the Test2 action if the result is negative and treat if the first or second test is positive [1]. With two tests, 99.96% of the patients with the disease are treated but 19% of the healthy patients are also treated. The

[1] To simplify our examples, we assume that repeated actions are independent, but in general this is not true.

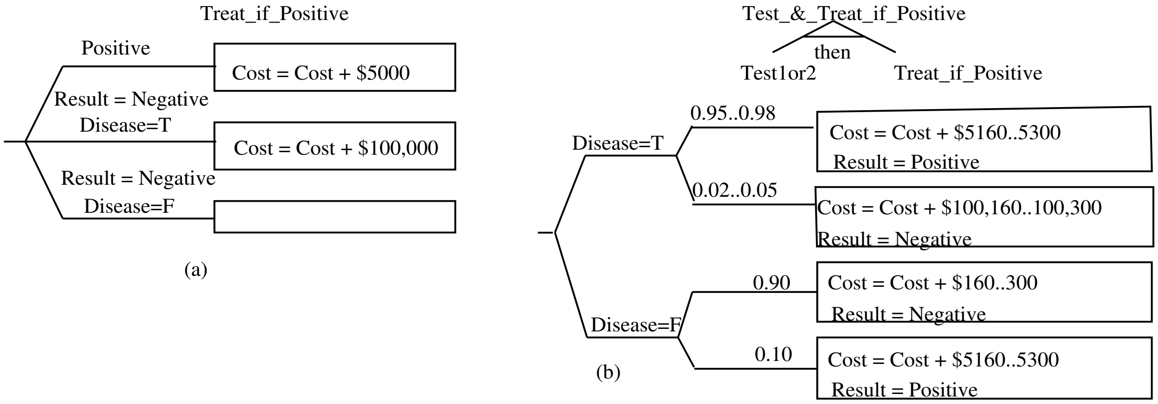


Figure 2: The sequence of a Test10r2 action and Treat_if_Positive action (a) are abstracted into a Test_&_Treat_if_Positive macro action (b).

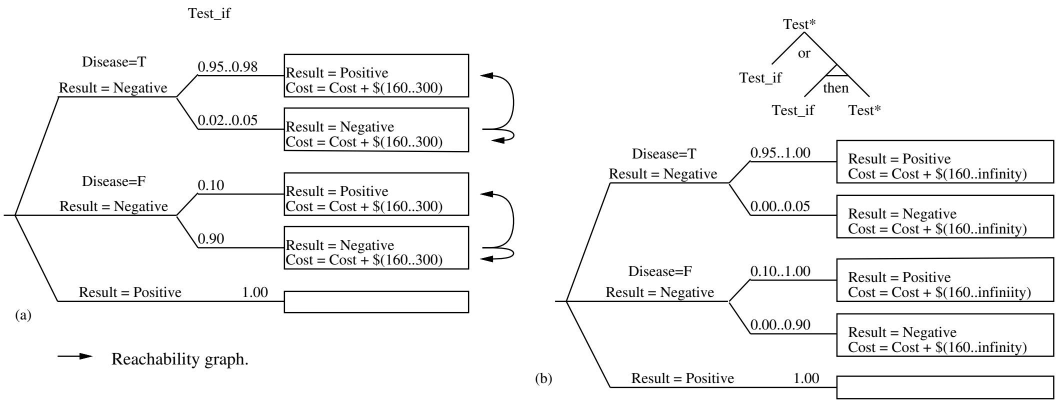


Figure 3: The Test_if action applies a Test action if there is no previous positive test result. Test*, a repeated action, is either a single Test_if action or a sequence of a Test_if action followed by a Test* action. The arrows show the reachability graph.

expected cost of using two tests is $3432, so using two tests is better than using one test.

If two tests are better than one, then are three tests better than two? What about four or more tests? What about using different combinations of Test1 and Test2? In this example, each time the test is repeated, the likelihood that patients with the disease will be treated goes up. However, the cost of the testing and the likelihood that healthy patients will be treated also goes up. In the limit, if the test is repeated until there is a positive result, everyone will be treated. The expected cost of the plan is the cost of treatment plus the cost of the expected number of tests. Clearly, the option to always treat is better than this and the sequence of tests should be terminated when the marginal gain is less than the marginal cost. The problem we address is determining the number and types of tests to perform.

Action Abstraction with Recursion

In a decision-theoretic refinement planner, the set of possible plans is represented in an action hierarchy. In the original version of the DRIPS planner, the space of plans represented in the abstraction hierarchy was finite, although arbitrarily large. The abstraction hierarchy did not allow recursion, a natural method of representing loops. In the this section we give an example using a recursive action.

Figure 3a defines a conditional action that applies a test unless a previous test result was positive. Figure 3b gives the recursive definition of the Test* action that includes one or more conditional tests. The effects of the Test* action give the range of possible outcomes. In the limit, the probability of detecting the disease is one, but the probability of a false positive is also one. The cost can range from the cost of one test to infinity for an infinite number of tests. These conditional effects are used by the planner to evaluate the bounds on the expected utility of a plan with the Test* action.

The planner begins with a plan that consists of the Test* action followed by the Treat_if_Positive action. This abstract plan represents all possible sequences of one or more tests followed by a treatment if one of the tests is positive. The planner proceeds by expanding the Test* action to create two plans: one with a single test and the other with two or more tests. Each plan is evaluated to determine the range of expected utility. If either plan is dominated, it is pruned from the search space. Expansion of the Test* action continues until the upper bound on the plan with the recursive action falls below the lower bound on one of the plans with a finite number of tests. The planner terminates when a dominant plan is found. In this example, the planner terminates with a plan that has two test actions, a Test1 action followed by a Test2 action, which is the optimal plan with an expected cost of $3325.

Method for Encoding Loops

A loop is a sequence of actions that is repeated one or more times until some condition is reached. The body of the loop is represented by a conditional action that applies the sequence of actions unless the condition holds. If the condition holds, the action has no effect. Test_if is an example of such an action that applies a test unless a previous test result was positive. A sequence of one or more repetitions of the loop is represented as an instance action that expands into a single action or into a macro action that, in turn, expands into a single action followed by the repeated action (Figure 3b).

The conditional effects of the repeated action can be derived from the effects of the action that form the body of the loop. We sketch here a method for doing this using a reachability graph. A reachability graph is created by connecting a directed arc from each effect in the conditional action to every effect that could be reached in the next iteration of the loop, except for the null effect used to terminate the loop. The arrows in figure 3a give the reachability graph for the test_if action. Each of the effects with a negative result can loop back to itself or to the effect with the positive result on the next iteration of the loop. Effects with a positive result will terminate the loop on the next iteration and have no outgoing arrow.

To determine the bounds on the probabilities, we begin by assigning each node in the reachability graph the probability of the effect in the conditional action. The reachability graph shows how this probability mass can flow as the loop is unrolled. The upper bound on probability is the upper bound on the sum of the probabilities that could flow into a node, plus its initial value. For example, the bound for the first condition in Test* is $[0.95 \ldots 0.98] + [0.02 \ldots 0.05] = [0.97 \ldots 1.03] \Rightarrow 1.00$. The lower bound will be zero if there is an edge leading from the node, otherwise the lower bound is the initial value. In the example, the lower bound on probability for the same conditions is the initial value, $[0.95 \ldots 0.98] \Rightarrow 0.95$.

Attribute values are also assigned using the reachability graph. If an effect in the conditional action assigns an attribute a specific value, then the effect in the repeated action assigns the attribute the same value (Result = Positive for example). Otherwise, the effect on the attribute must include the effects along any path that reaches the effects node. In the case of a loop, we need to find the bounds for a single cycle of the loop and the bounds as the number of cycles of the loop approaches infinity. For the self loop in the case of a negative result, $cost_{i+1} = cost_i + [160..300]$. The number of iterations ranges from 1 to ∞ and the bounds on cost are the bounds on the recursive equation. Since the recurrence equation is monotonically increasing for increasing i, the minimum value occurs for $i = 1$ and the maximum value occurs at $i = \infty$. Taking the limit as $i \to \infty$ gives $cost_\infty = cost_0 + \$\infty$. The resulting conditional effect is $cost = cost_0 + \$[160 \ldots \infty]$

Our method of encoding loops ensures that the repeated action contains at least one iteration of the loop. The effects of the repeated action then do not have to include the effects of zero iterations of the loop. This provides tighter bounds on the range of effects and allows plans with loops to be pruned with fewer unrollings of the loop. Since the cost of projecting a plan is exponential in the length of the plan, saving one loop unrolling leads to significant performance improvements.

In related work, Ngo and Haddawy give a constructive method for generating the conditional effects for a loop repeated up to N times (Ngo & Haddawy 1995). The method reformulates the conditional effects of the repeated action and can lead to tighter bounds in some cases. Their method could be extended to work for infinite loops by using limits as N approaches infinity.

Finding Optimal Plans with Finite Length

In the simple medical domain, the optimal plan is a finite sequence of actions. In general, the value of repeated information gathering actions, like a medical test, is limited by the value of perfect information. The planner will expand the recursive action until the value of information from the test is less than the cost of the test, and begins to lower the upper bound on expected utility. When the upper expected utility bound on the plan with the infinite action falls below the lower bound on the expected utility of another plan, it can be pruned. This method of pruning constitutes a proof that the infinite plan is not an optimal plan.

As with any proof method, we must be concerned about soundness and completeness. The result is sound if optimal plans are not pruned, while it is complete if all the suboptimal plans are pruned and the planner terminates. The soundness of the planning method is discussed in (Doan & Haddawy 1995b). In this section, we give sufficient conditions for ensuring that optimal plans are finite and sufficient conditions for the DRIPS planner to find the optimal plans and terminate.

Since the algorithm terminates only when the set of potentially optimal plans can no longer be expanded, the algorithm will never terminate when the optimal plan consists of an infinite sequence of actions. Consider, for example, a domain where a robot is trying to pick up a cup. Suppose there is a single pick_up_cup action that succeeds with probability 0.5 and that the value of holding the cup is 1 and the cost of trying to pick it up is 0.1. The expected value of trying to pick up the cup once is 0.4. If the action fails, we are back in the original state. Assuming independence, the expected utility of two attempts to pick up the cup is 0.6. As the number of pick up attempts is increased, the expected utility asymptotically approaches the value of the infinite plan, 0.8. The planner can not recognize that the optimal plan is infinite and will attempt to unroll the loop indefinitely.

Fortunately, in many domains the optimal plan is not infinite. We can ensure that this is the case by imposing conditions on the domain and conditions on the domain description that allow the planner to determine when infinite plans can be pruned. The optimal plan will be finite if, after some fixed number of unrollings, the marginal utility of unrolling the loop once more is negative and, for each additional unrolling, the marginal utility is non-positive. Under these conditions, the expected utility of the infinite plan will converge to minus infinity, or some sub-optimal finite value. The same condition can be imposed on the upper bound on expected utilities for plans generated from the domain description, including plans with abstract actions. This ensures that plans with infinite recursion can be pruned after some number of unrollings of each loop. For the previous cup example, we can ensure termination by making the utility function time dependent and have the value of picking up the cup become zero after five minutes.

There are many possible restrictions on the encoding of a domain and a utility function that can enforce the sufficient conditions necessary to ensure finite length optimal plans. These can include restrictions on the form of the utility function and restrictions on the actions in the domain. One method is to have each iteration of a loop consume some finite amount of a limited resource. Plans that violate the resource limit are assigned low utility. Ngo and Haddawy have proposed using this method (Ngo & Haddawy 1995). Resource limits are used to calculate a limit on useful loop iterations, and the domain is encoded to only allow that many iterations of the loop. Williamson and Hanks use a more restrictive condition that limits all resources and allows only deadline goals (Williamson & Hanks 1994). They use these restrictions to do optimal planning with a partial order planner.

Handling Infinite Length Optimal Plans

For some domain descriptions, the optimal plan may include an infinite loop. The optimal plan for the pick_up_cup example is an infinite sequence of attempts to pick up a cup. In practice, it is impractical to attempt to execute a plan with an infinite loop. The appearance of an infinite loop in an optimal plan is likely caused by limited fidelity in the domain model or the utility function. On each iteration of the pick_up_cup action, the marginal utility is cut in half. At some point, using the difference in expected utility for a finite length plan and an infinite length plan will not be justified by the precision of the probability and attribute value estimates used to calculate the expected value. Independence assumptions also limit model fidelity. In the cup example, the probability of success for subsequent attempts to pick up the cup may not be independent. An unmodeled attribute, like the cup being too big, may prevent the cup from ever being grasped. Failure on the first attempt should serve as evidence to lower the probability of success on subsequent trials, leading to an optimal plan with a finite number of repetitions.

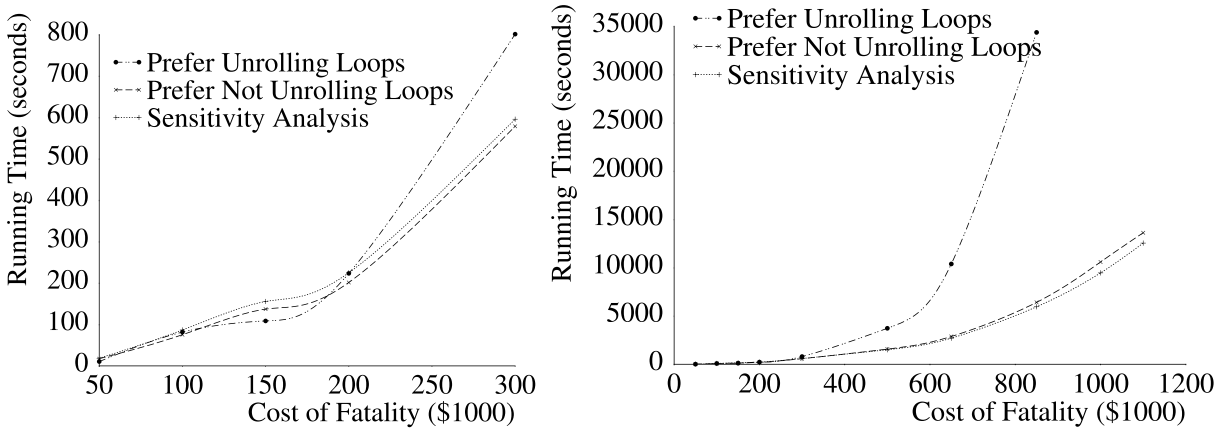
To keep the DRIPS planner from trying to enumerate all the actions in an infinite optimal plan, we add two additional pruning methods for plans with loops. We prune infinite plans when the difference in expected utility for a finite plan and the infinite plan is less than the accuracy of the numbers used to calculate the expected utility. This method also helps to deal with problems due to numerical roundoff errors. We also prune infinite plans when the difference in expected utility is not justified by the computation time needed to unroll the loop one more time. For offline planning, we use a user supplied function to map computation time to cost (negative utility). We require the cost function to monotonically increase with increased time. Since each unrolling of a loop lengthens the plan to be evaluated and plan evaluation time is exponential in the length of the plan, this condition is sufficient to ensure planner termination. For online planning, the the utility function could be used determine the cost of delaying execution.

Search Control for Infinite Plan Spaces

With the introduction of loops and the resulting infinite plan spaces, search control becomes even more crucial for efficient planning. In previous work, we have shown that an optimal policy exists for choosing which plan to expand next and that search efficiency depends only on which action within this plan is selected for expansion (Haddawy, Doan, & Goodwin 1995). In this section, we discuss the factors that affect search efficiency and describe extensions to our sensitivity analysis-based search control method(Goodwin 1994) that allow it to deal with loops. In the next section, we compare three search heuristics using a large medical domain with loops.

A key question for search control in domains with loops is when to unroll the loop. Expanding the recursive action preferentially can result in having to evaluate very long plans. The loop has to be unrolled until the falling marginal utility of each unrolling causes the upper bound on expected utility to drop below the lower bound for some finite length plan. With abstract actions, the range of expected utility for the finite length plans can be large, leading to excessive unrolling. Evaluating these long plans is expensive since the cost of projection is exponential in the length of the plan. On the other hand, expanding recursive actions only when all other actions have been fully expanded creates tighter bounds on the expected utility of finite length plans but creates many plans with recursive actions that may have to have their loops unrolled.

Since it is the upper bound on expected utility that is used to prune a plan, a good strategy is to select expansions that will have the largest effect on reducing the upper bound, taking into account the amount of computation needed. Approximating this strategy is the idea behind sensitivity analysis-based search control. The domain description of actions in the plan and the utility function are used by the planner to determine the maximum change in the upper bound for fully expanding an abstract action into primitive actions. This value is then divided by an estimate



Execution Time Versus Cost of Fatality

Execution Time Versus Cost of Fatality

Figure 4: Empirical results showing running time versus cost of fatality for three search control methods and for two ranges of the cost of fatality. The graph on the left is an enlargement of part of the graph on the right.

of the work needed to fully expand the action. This ratio is an estimate of the rate of lowering the upper bound on expected utility per unit of computation. The action with the highest ratio is chosen for expansion.

To use the sensitivity method for recursive actions, we cannot calculate the maximum possible change in expected utility from fully expanding the action and then divide it by the work needed to fully expand the action. Both numbers could be infinite. Instead, we estimate the rate by considering only a single unrolling of the loop. A bound on the change in expected utility can be calculated using the change in attribute values for a single iteration of the loop. The work needed is two plan evaluations, one with one more iteration and the other with one or more iterations. Again, the plan with the highest expected ratio of bounds lowering to work is chosen for expansion.

In previous work, we had used the number of plan evaluations as an estimate of the amount of computation. This is appropriate when each plan evaluation takes approximately the same amount of computation. With loops, the length of plans can vary significantly. Since plan evaluation is exponential in the length of the plan, the number of plan evaluations is not a good measure of computation. Instead, we use the structure of the action being expanded to weight the number of plan evaluations by relative computation cost. Instance actions that expand into a single action do not lengthen the plan and have a relative cost of one. Sequence actions lengthen the plan by the number of actions in the sequence minus one. Their relative weight is the average branching factor raised to the increase in length, $(\overline{b})^{\triangle length}$. In our experience, using the new measure leads to more plan evaluations, but less computation on average.

Empirical Results

The DRIPS planner has been used to analyze treatment policies for deep venous thrombosis (DVT) (Haddawy, Kahn, & Doan 1995). The domain model is taken from a study in the medical literature that evaluated treatment policies, including repeated tests (Hillner, Philbrick, & Becker 1992). The domain consists of a model of how the disease can progress over time and the actions available to the doctor which include a number of tests and a treatment. The planning objective is to create a testing and treatment policy with the highest expected utility. The original encoding of the domain for the DRIPS planner allowed up to three tests before a decision to treat was made. We have extended the domain to use loops so that the planner can determine how many tests are useful.

We have run a series of experiments using the modified DVT domain to compare the performance of three search control strategies. In the experiments, we vary the cost of fatality. For a very low cost of fatality, the optimal plan is not to test or treat anyone. As the cost of fatality rises, the optimal plan includes more tests to improve the likelihood of detecting the disease and treats the patient when the test results are positive. Adding more tests increases the probability that the patients will suffer side effects from the tests, which can lead to death in some cases. A utility function is used to weigh the relative benefits.

The DRIPS planner allows the user to supply search control information by assigning relative priorities for expanding to each abstract action. To evaluate the performance of our sensitivity analysis method, we have compared it to two hand-tuned priority schemes. One priority scheme was optimized to produce good performance for a low cost of fatality and generally prefers unrolling loops. The other was optimized for higher costs of fatality and prefers not unrolling loops.

In the DVT domain, deciding when to unroll the test loops can have a significant affect on performance, especially for harder problems. For low costs of fatality, preferentially unrolling the loop quickly shows that more tests are not utile, and the plan with the loop is quickly pruned. However, these problems are easily solved and the time difference is minimal (figure 4). For higher costs of fatality, preferentially expanding other abstract actions leads to evaluating shorter plans on average, which gives much better time performance. The sensitivity analysis-based search control tends to follow the best strategy at each extreme. At low costs of fatality, all three methods evaluate few plans (between 9 and 13). There are relatively few choice points and making a bad decision does not lead to excessive computation. The overhead of doing the sensitivity analysis calculation can not be fully compensated for by making better decisions. At higher costs of fatality, there are significant numbers of choice points and making good decisions leads to significant speed improvements. As the cost of fatality increases, the performance of the sensitivity analysis-based method approaches and eventually out performs the priority method optimized for high costs of fatality. In these harder problems, the improvement in decision quality more than compensates for the cost of the sensitivity analysis. The sensitivity analysis-based method adapts to the specific problem to provide efficient search control over the range of planning problems considered. This is done without requiring the user to supply additional search control information.

Discussion

This paper describes a decision-theoretic refinement planner extended to handle loops of repeated actions. We gave conditions under which the optimal plan will be finite and conditions under which the planner will find the optimal plan and terminate. To deal with infinite optimal plans, we have introduced two additional criteria for pruning plans with infinite loops: prune when the expected cost of computation exceeds any possible benefit and when the difference in expected utility is less than the accuracy of the values used in the domain description. We have demonstrated how our sensitivity analysis-based search control method provides effective search control, without burdening the user.

One area of concern for the DRIPS planner is that it requires the domain designer to decide which conditions to use for conditional actions. In our medical example, we allowed repeated tests if the result was negative. It may also be useful to allow repeated tests if the result was positive. This would be useful if the treatment was almost as bad as the disease. We are currently working on methods to allow a decision-theoretic refinement planner to automatically decide which conditions to use for loop termination.

Acknowledgements

This research was partially sponsored by NASA under grants NAGW-3863 and NAGW-1175. I wish to thank Reid Simmons for many useful discussions and comments on this paper. I also wish to thank Peter Haddawy for making the DRIPS planner available.

References

Blythe, J. 1994. Planning with external events. In de Mantras, R. L., and Poole, D., eds., *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, 94–101.

Doan, A., and Haddawy, P. 1995a. Generating macro operators for decision-theoretic planning. In *Working Notes of the AAAI Spring Symposium on Extending Theories of Action*.

Doan, A. H., and Haddawy, P. 1995b. Decision-theoretic refinement planning: Principles and application. Technical Report TR-95-01-01, Dept of Elect. Eng & Computer Science, University of Wisonsin-Milwaukee.

Goldman, R. P., and Boddy, M. S. 1994. Epsilon-safe planning. In de Mantras, R. L., and Poole, D., eds., *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, 253–261.

Goodwin, R. 1994. Reasoning about what to plan. In *Proceedings, Twelfth National Conference on Artificial Intelligence*. AAAI.

Haddawy, P., and Doan, A. 1994. Abstracting probabilistic actions. In de Mantras, R. L., and Poole, D., eds., *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, 270–277.

Haddawy, P.; Doan, A.; and Goodwin, R. 1995. Efficient decision-theoretic planning techniques. In Besnard, P., and Hanks, S., eds., *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, 229–236.

Haddawy, P.; Kahn, Jr, C.; and Doan, A. 1995. Decision-theoretic refinement planning in medical decision making: Management of acute deep venous thrombosis. *(submitted to Medical Decision Making)*. (submitted to *Medical Decision Making*).

Hillner, B. E.; Philbrick, J. T.; and Becker, D. M. 1992. Optimal management of suspected lower-extremity deep vein thrombosis: an evaluation with cost assessment of 24 management strategies. *Arch Intern Med* 152:165–175.

Kushmerick, N.; Hanks, S.; and Weld, D. 1994. An algorithm for probabilistic least-commentment planning. In *Proceedings, Twelfth National Conference on Artificial Intelligence*, 1073–1078. AAAI.

McDermott, D. 1982. A temporal logic for reasoning about processes and plans. *Cognitive Science* 6(2):101–155.

Musliner, D. J. 1994. Using abstraction and nondeterminism to plan reactive loops. In *Proceedings, Twelfth National Conference on Artificial Intelligence*, 1036–1041. AAAI.

Ngo, L., and Haddawy, P. 1995. Representing iterative loops for decision-theoretic planning (preliminary report). In *Working Notes of the AAAI Spring Symposium on Extending Theories of Action*.

Williamson, M., and Hanks, S. 1994. Optimal planning with a goal–directed utility model. In Hammond, K., ed., *Proceedings of the Second International Conference (AIPS94)*, 176–181. Artificial Intelligence Planning Systems.